# Corrigé TP13

Le corrigé est présenté en Ocaml. Tout s'adapte en C.

5. A priori, la démonstration que vous avez vue en mathématiques doit être de l'acabit suivant. Soit $n \in \mathbb{P}$. Montrons par récurrence sur $a \in \mathbb{N}$ que $a^n \equiv a \bmod n$.

   L'initialisation est immédiate. Pour l'hérédité, il suffit de se rappeler que la formule du binôme de Newton assure que :
$$(a+1)^n = \sum_{k=0}^{n} \binom{n}{k} a^k \equiv 1 + a^n \bmod n$$
   puisque tous les autres coefficients de la somme font intervenir un coefficient binomial divisible par $n$ en raison du fait que ce dernier est premier (je vous laisse refaire cette preuve). Comme l'hypothèse de récurrence assure que $a^n \equiv a \bmod n$, on conclut.

   On déduit de la récurrence précédente que pour tout $a \in \mathbb{N}$, $n$ divise $a(a^{n-1}-1)$. Lorsque $a$ est premier avec $n$, le théorème de Gauss assure alors que $n$ divise $a^{n-1}-1$, c'est-à-dire que $a^{n-1} \equiv 1 \bmod n$. Voilà le petit théorème de Fermat démontré.

6. On fait attention à ne pas répéter de calcul déjà effectué et à bien réduire modulo $p$ au fur et à mesure. Si $P_n$ est le nombre de produits modulaires effectués par expo_modulaire lorsque l'exposant est $n$, du code découle la relation de récurrence suivante :
$$P_n = P_{n/2} + O(1)$$
   On obtient bien un nombre de produits modulaires en $O(\log n)$ comme exigé.

```
let rec expo_modulaire (x:int) (n:int) (p:int) :int =
  match n with
    |0 -> 1
    |n when (n mod 2 = 0) -> let moitie = expo_modulaire x (n/2) p in
                             (moitie*moitie) mod p
    |_ -> let moitie = expo_modulaire x (n/2) p in (moitie*moitie*x) mod p
```

7. On reprend la partie 1 pour implémenter un crible d'Eratosthène en Ocaml.

```
let crible_eratosthene (n:int) :bool array =
  let p = Array.make (n+1) true in
  p.(0) <- false;
  p.(1) <- false;
  for d = 2 to n do
    if p.(d) then
      begin
        let m = ref (d*d) in
        while !m <= n do
          p.(!m) <- false;
          m := !m + d
        done
      end
  done;
  p
```

Pour écrire la fonction poulet, on commence par déterminer quels sont les premiers inférieurs à l'entrée $n$ puis pour chaque entier $i$ inférieur à $n$, on vérifie si $i$ passe le test de Fermat pour 2 tout en étant non premier. Comme on va tester la primalité de tous les entiers inférieurs à $n$, on préfère calculer cette information une bonne fois pour toute avec le crible plutôt que de tester indépendamment la primalité de chacun avec l'algorithme naïf.

```
let poulet (n:int) :unit =
  let p = crible_eratosthene n in
  for i = 2 to n do
    if ((expo_modulaire 2 (i-1) i) = 1) && (not p.(i))
    then Printf.printf "%d est un nombre de Poulet\n" i
  done
```

Les nombres de Poulet inférieurs à 1000 sont 341, 561 (qui est aussi de Carmichael), 645.

8. L'algorithme est le suivant : étant donné $n$, tirer un entier $a \in [\![1, n-1]\!]$, tester si $a^{n-1} \equiv 1 \bmod n$ :

   - Si ce n'est pas le cas, alors il est certain que $n$ n'est pas premier : on renvoit faux.
   - Sinon, $n$ est probablement premier : on renvoit vrai.

   Cet algorithme peut répondre incorrectement mais l'erreur est bien unilatérale :

   - Si $n$ est premier alors l'algorithme ne peut que renvoyer vrai : il n'y a pas de faux négatif.
   - Si $n$ n'est pas premier, il peut renvoyer faux comme vrai.

   Le temps de calcul sur une entrée est indépendant des choix aléatoires. L'implémentation suit :

```
let est_premier_fermat (n:int) :bool =
  let a = (Random.int (n-1)) +1 in
  (expo_modulaire a (n-1) n) = 1
```

9. Soit $n \notin \mathbb{P}$ un nombre qui n'est pas de Carmichael. On vérifie facilement que $G = \{a \in [\![1, n-1]\!] \mid a^{n-1} \equiv 1 \bmod n\}$ est un sous groupe de $(\mathbb{Z}/n\mathbb{Z})^\star$. D'après le théorème de Lagrange, $|G|$ divise donc $|(\mathbb{Z}/n\mathbb{Z})^\star|$.

   Mais on sait de plus que $G$ est un sous-groupe strict de $(\mathbb{Z}/n\mathbb{Z})^\star$ puisqu'il n'est pas de Carmichael. On en déduit que $|G| \leq |(\mathbb{Z}/n\mathbb{Z})^\star|/2 = (n-1)/2$ puisque tous les éléments de $\mathbb{Z}/n\mathbb{Z}$ sauf 0 sont inversibles lorsque $n$ est premier. Ainsi, la probabilité $p$ de faux positif de est_premier_fermat sur $n$ — qui est égale à la probabilité de tirer un élément de $G$ dans $[\![1, n-1]\!]$ — vérifie :

$$p = \frac{|G|}{|[\![1, n-1]\!]|} \leq \frac{n-1}{2(n-1)} \leq \frac{1}{2}$$

10. On cherche à amplifier l'algorithme Monte Carlo de la question 8 afin d'obtenir une probabilité de faux positif aussi petite que voulue. Notons $B$ l'algorithme consistant à effectuer $k$ itérations indépendantes de est_premier_fermat et renvoyant true si et seulement toutes les itérations renvoient true.

    Si $n$ est composé, la probabilité que $B$ renvoie true malgré tout est égale à la probabilité que $B$ renvoie true et que $n$ soit de Carmichael plus la probabilité que $B$ renvoie true et que $n$ ne soit pas de Carmichael. Le premier terme de cette somme est négligé par l'énoncé et le deuxième est inférieur à $1/2^k$ d'après la question 9. On cherche donc $k$ tel que cette quantité soit inférieure à $10^{-20}$ : le plus petit $k$ convenable est 67 d'où l'algorithme suivant :

```ocaml
let est_presque_premier_fermat (n:int) :bool =
  let res = ref true in
  for i = 0 to 66 do
    res := !res && est_premier_fermat n
  done;
  !res
```

11. Soit $n$ un nombre premier impair et $a \in [\![1, n-1]\!]$. Le petit théorème de Fermat assure que $a^{n-1} \equiv 1 \bmod n$ donc que $a^{m\times 2^s} \equiv 1 \bmod n$ ce qui se réécrit $(a^{m\times 2^{s-1}})^2 \equiv 1 \bmod n$.

    Cette dernière égalité signifie que $a^{m\times 2^{s-1}}$ est une racine de 1 modulo $n$ donc est égal à $-1$ ou 1. Dans le premier cas, la propriété (2) est vérifiée, dans le second cas, $a^{m\times 2^{s-1}} \equiv 1 \bmod n$. Si $s = 1$ la propriété (1) est vérifiée, sinon, la dernière congruence se réécrit $(a^{m\times 2^{s-2}})^2 \equiv 1 \bmod n$.

    On peut donc réitérer le raisonnement ci-dessus : au fil des factorisations soit on va trouver $d \in [\![1, s-1]\!]$ tel que $a^{m\times 2^d} \equiv -1 \bmod n$, soit on aura finalement $(a^m)^2 \equiv 1 \bmod n$ et dans ce cas $a^m$ est une racine de 1 modulo $n$ donc est congru soit à 1 soit à $-1$ : dans le premier cas, (1) est vérifiée, dans le second (2) est vérifiée. Par conséquent, pour tout $n$ impair premier et tout $a \in [\![1, n-1]\!]$, on a toujours soit (1) soit (2) qui est vérifiée.

    *Remarque : L'indication de l'énoncé provient du fait que si $n$ est premier alors le polynôme $X^2 - 1 \in \mathbb{Z}/n\mathbb{Z}[X]$ est à coefficients dans un corps donc admet moins de deux racines dans $\mathbb{Z}/n\mathbb{Z}$ : comme 1 et $-1$ sont racines, la propriété est démontrée.*

12. On peut déjà gérer à part le cas des entiers pairs : on renvoie vrai pour 2 et faux pour tous les autres. Pour les entiers $n$ impairs, le principe est le même que pour l'algorithme de Fermat : tirer un entier $a \in [\![1, n-1]\!]$, vérifier si c'est un témoin de Miller-Rabin : si oui, il est certain que $n$ n'est pas premier par contraposée de R et on renvoie faux, si non, $n$ est probablement premier et on renvoie vrai. Cette construction donne naissance à un algorithme qu'on appelle $A$.

    Avec cette construction, il est impossible d'avoir un faux négatif : si $n$ est véritablement premier il n'a pas de témoin de Miller-Rabin donc $A$ ne peut que renvoyer vrai. Si $n$ est composé, la probabilité que $A$ renvoie vrai est égale à :

    $$\begin{aligned} & P(A \text{ renvoie vrai et } n \text{ est pair}) + P(A \text{ renvoie vrai et } n \text{ est impair}) \\ =& 0 + P(A \text{ renvoie vrai et } n \text{ est impair}) \quad \text{car dans le cas des entiers pairs, } A \text{ est toujours correct} \\ =& P(\text{l'entier } a \text{ choisi n'est pas un témoin pour } n) \\ \leq& \frac{1}{4} \quad \text{d'après le théorème de Rabin} \end{aligned}$$

    La probabilité de faux positif pour $A$ est donc inférieure à 1/4.

13. On suit les indications de l'énoncé en construisant `verifie_critere` : elle prend en entrée $n, m, s, a$ tels que $n - 1 = 2^s \times m$ et renvoie `true` si et seulement si $a$ vérifie l'une des propriétés parmi (1) et (2).

    La fonction auxiliaire `puissances_2` teste spécifiquement la propriété (2). Elle calcule successivement $a^m \bmod n$, $a^{2m} \bmod n$, $a^{2^2 m} \bmod n$ ; pour calculer le $i$-ème terme de cette suite, il suffit de mettre le $(i-1)$-ème au carré. Elle vérifie si l'un de ces termes est congru à $-1$ modulo $n$ : si oui, (2) est vérifiée et si la puissance de 2 dépasse $s-1$, elle ne l'est pas.

```
let verifie_critere (n:int) (m:int) (s:int) (a:int) :bool =
  let rec puissances_2 (a_puissance:int) (exposant:int) :bool =
    match a_puissance, exposant with
      |p, _ when  (p = n-1) -> true
      |_, d when (d > s-1) -> false
      |p, d -> puissances_2 (expo_modulaire p 2 n) (exposant+1)
  in ((expo_modulaire a m n) = 1) || (puissances_2 (expo_modulaire a m n) 0)

let temoin n m s a = not (verifie_critere n m s a)
```

La fonction temoin renvoie true si et seulement si $a$ est témoin de Miller-Rabin pour $n$ tel que $n-1 = 2^s \times m$ : il s'agit juste de nier le résultat obtenu avec verifie_critere.

*Remarque : Attention, au fonctionnement de mod en Ocaml : p mod q renvoie le reste dans la division euclidienne de $p$ par $q$. En particulier 55 mod 56 n'est pas égal à $-1$ selon Ocaml d'où la gestion du premier cas de la fonction puissances_2. Remarque similaire en C.*

Une fois la fonction temoin implémentée, on construit une fonction decomposition permettant de calculer les entiers $s$ et $m$ tels que $n-1 = 2^s \times m$ lorsque $n$ est impair :

```
let rec decomposition (n:int) :int*int = match n with
  |0 -> (0,0)
  |n when (n mod 2 = 1) -> (0,n)
  |n -> let (s,m) = decomposition (n/2) in (s+1,m)
```

Il ne reste plus qu'à itérer $k$ fois l'algorithme décrit à la question 12 pour obtenir (toujours d'après la question 12) une fonction convenable dont la probabilité d'échec est inférieure à $(1/4)^k$ :

```
let est_presque_premier_mr (n:int) (k:int) = match n with
    |0 |1 -> false
    |2 -> true
    |n when n mod 2 = 0 -> false
    |_ -> let (s,m) = decomposition (n-1) in
          let res = ref true in
          for i = 1 to k do
            let a = (Random.int (n-1)) +1 in
            res := !res && (not (temoin n m s a))
          done;
          !res
```